Proving with the Help of Property-Based Testing

Exposing the QuickCheck Interface in Agda for AGDA2HS

Maurits L. C. Sloof[1] (✉)

Supervisors: Jesper G. H. Cockx[1], **Nathaniel Burke**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Maurits Lourens Christiaan Sloof
Final project course: CSE3000 Research Project
Thesis committee: Dr. Jesper G. H. Cockx, Nathaniel Burke, Dr. Annibale Panichella

Abstract

Testing constitutes an important part of software development for many engineers. AGDA2HS, a transpiler from Agda to Haskell, currently does not provide such testing functionality. Agda enables developers to express and formally verify properties directly within the source code, whereas Haskell benefits from an extensive ecosystem of tools and developers. By transpiling verified Agda code to Haskell, AGDA2HS offers the potential to combine the advantages of both languages. However, constructing formal proofs in Agda is challenging, as it requires the same creativity and insight needed for mathematical reasoning. Consequently, a proposition with a bug may be mistakenly seen as difficult to prove. Property-based testing can provide confidence in the correctness of a proposition before a proof is completed. This paper presents these testing functionalities in AGDA2HS by porting QuickCheck to Agda. Furthermore, correspondence proofs are introduced, which enable programmers to show formally that a test asserts the same property as a proposition.

1 Introduction

Correctness proofs can provide strong guarantees for software systems in which failures may have severe consequences. Examples include compilers [1], autopilots [2] and software controlling (medical) radiation emitters [3].

However, writing these proofs can be substantially more difficult than writing the software itself. To address this issue, AGDA2HS was introduced as a transpiler from Agda, a functional programming language in which correctness proofs can be written natively, to Haskell, a general functional programming language [4]. Nevertheless, developing verified Agda code remains challenging, as a proposition with a bug—either in the proposition itself or in its translation to Agda—cannot be proven, but identifying such a bug can take considerable time.

Hence, I introduce software testing to AGDA2HS, enabling programmers to be reasonably confident in the correctness of a proposition before attempting to prove it formally. This is achieved by porting QuickCheck to AGDA2HS and extending it with additional functionality to prove a correspondence between a test and a proposition.

1.1 Agda and AGDA2HS

Agda is a programming language based on a dependent type system [5]. In a dependent type system, types may depend on values. In combination with the Curry-Howard correspondence [6], Agda can be used as a proof assistant. By 'rephrasing' a proposition into a function signature, an implementation of that function acts as its proof. For this proof to be correct, the fact that Agda functions are total and thus always return values in their codomain is crucial. Otherwise a function returning any arbitrary type can be constructed by always throwing an exception. The creative part of the proof is the implementation of the function, which serves as constructive evidence that such a function exists.

AGDA2HS [4] is a program that transpiles Agda code to Haskell code. By making this translation possible, programmers can produce Haskell code with the guarantees of Agda. This benefits Haskell developers by creating verified implementations and Agda users by extending the applicability of Agda to broader deployment contexts.

1.2 Contributions

The purpose of this thesis is to explore the feasibility of adding a testing framework, specifically one based on the Haskell QuickCheck library [7], to AGDA2HS to allow for the testing of propositions. As stated above, developing Agda code is difficult, especially for programmers not familiar with the underlying type-theoretic foundations [8]. In addition, the difficulty of constructing formal proofs further complicates mathematically rigorously verifying Agda code. For this paper, I ported QuickCheck to Agda for AGDA2HS, answering the following research question:

> How can the interface of the Haskell QuickCheck library and its implicit properties be exposed in Agda for use with AGDA2HS?

In this paper, an approach to porting QuickCheck is presented. First, the QuickCheck interface was analysed to identify the functionality required for proposition testing in AGDA2HS. Subsequently, the selected functionality was exposed through Agda bindings and extended with correspondence proofs and helper functions. Finally, the port is evaluated through an analysis of its capabilities and limitations.

1.3 Overview

This thesis first further explains Agda, QuickCheck and AGDA2HS in section 2. Then, the approach to adding testing to AGDA2HS is outlined in section 3. Subsequently, the results and limitations of this implementation are discussed in section 4. Section 5 presents the ethical principles on which this thesis was written. Then, related work of introducing testing functionality to other proof assistants is summarised in section 6, followed by a comparison between this port and the related work in section 7. Finally, section 8 provides the conclusions and suggestions for future work on this port.

2 Background

In this section, the background of this paper is discussed. Since this paper contains both Agda and Haskell code, which are occasionally difficult to distinguish, all Haskell code in this paper has a grey-tinted background. First, there is a brief introduction to Haskell in subsection 2.1 followed by a more extensive description of QuickCheck in subsection 2.2. Then, in subsection 2.3, the fundamentals of Agda are explained. Finally, there is an introduction to AGDA2HS in subsection 2.4.

2.1 Haskell

Haskell is a general-purpose strongly typed, purely functional programming language using lazy evaluation. Since Haskell

is a purely functional programming language, it is often easier to reason about properties of algorithms and data structures written in Haskell than in imperative programming languages, such as C, Python and Rust [4; 9]. However, proofs concerning Haskell code are written 'on paper' and not in the Haskell source code itself, allowing for desynchronisations between the proof and the code. These desynchronisations could occur by updating the code after writing the proof or by making a copying error. In addition to these desynchronisations, there is also the possibility that the paper proof contains an error.

2.2 QuickCheck

To test Haskell programs, Claessen *et al.* introduced the Property-Based Testing (PBT) library QuickCheck [7]. Using QuickCheck, properties of programs can be tested by checking whether they hold for randomly generated data. After generating many such data points—by default, a hundred—for which the property holds, a programmer may be reasonably confident that their code is correct. Conversely, a failing test means that the property does not hold. Such a failing test also produces a counterexample, which helps programmers find bugs residing in the code or the property specification.

Test inputs are generated by a generator for the corresponding type. A generator of type `a` is a function wrapped in a Haskell `newtype Gen a` which, given a random number generator (RNG) of type `QCGen` and a size argument of type `Int`, produces a value of type `a` [10].

The size parameter serves as an upper bound on the complexity of the value generated by the generator, for instance, the maximum length of a list. QuickCheck varies this parameter during the generation of data points, allowing for a diverse set of test inputs. It also allows programmers to influence the size of the data points that QuickCheck generates. This is particularly important for recursive generators, as it can prevent excessive computation by reducing the size parameter in recursive calls. For example, if a binary tree is generated while ignoring the size parameter but with a 25% chance of generating a leaf node on every (recursive) call, then two out of three generation attempts will result in infinite trees[1] and thus, if it consumes the entire tree, a non-terminating program. This is the case, as every node, on average, is likely to produce more than one child node.

It is up to the programmer creating the generator to decide how to interpret the size argument, but it will not take on a negative value. As stated above, it can be an upper bound on the length of a list, but it can also be the exact size of a list. Additionally, there could be multiple definitions of 'size', such as the depth versus the total number of nodes in a tree. Finally, some generators ignore the size entirely when it is not relevant to the generated type, such as for boolean values ($\top$ or $\bot$) [12].

With these generators, properties about their types can be tested. For example, the following test asserts that adding zero to an integer preserves its value:

```
prop_add_identity :: Int -> Property
prop_add_identity i = i + 0 === i
```

The operator '`===`' in QuickCheck differs from Haskell's standard equality operator '`==`' in that it returns a `Property` rather than a `Bool`. Returning a `Property` is similar to returning a `Bool` in the testing sense—QuickCheck even allows tests to return a boolean value—but a `Property` contains extra metadata about the comparison, allowing for more informative output on test failures.

While PBT is useful for general software development, it is inherently limited by the finite number of inputs explored, particularly for types with a large or infinite value space. Consequently, an obscure counterexample may be missed, such as a property that holds for all integers except 42. Furthermore, generators may have biases or blind spots, for which few to no values of that part of its supposed range will be generated. Those values will, in turn, also not be tested, leading to a misleading impression of correctness.

2.3 Agda

Agda [13] is a dependently typed, purely functional, total programming language and proof assistant. As stated previously, the syntax of Agda is similar to that of Haskell, although Agda supports Unicode characters in source code. Totality in Agda ensures that functions always return a value in their codomain, prohibiting exceptions and infinite loops. The language is based on the dependent type theory of Zhaohui Luo [14], which is related to Per Martin-Löf's type theory [15; 5].

Agda is built from three fundamental building blocks: `record`s, `data` types and functions. Records are analogous to structs in other languages, where all fields must be instantiated with a value. Data types resemble union types or Rust-style enums, in which a value is created by selecting one of the constructors. Finally, functions behave similar to Haskell's functions but are total.

Considering the Curry-Howard correspondence [6], Agda can be used as a proof assistant by representing a proposition as a function type, where implementations correspond to proofs. Such an implementation can only be constructed if the proposition encoded in the type holds. The Agda type checker therefore acts as a proof checker, rejecting programs with incorrect proofs. By writing programs and their related proofs in the same language, Agda avoids the desynchronisations that are possible in Haskell with pen-and-paper proofs.

Propositions are constructed by combining types within a function signature [16]. Similar to Haskell's "`A -> B`", "`A → B`" (or even the same 'non-unicode' "`A -> B`") denotes a function from `A` to `B`. In the context of the Curry-Howard correspondence, it also means "`A` *implies* `B`". Conjunction is encoded by product types such as a tuple ($\times$) and disjunction is encoded by sum types, such as `Either`. Truth corresponds to the unit type $\top$, while falsity corresponds to the empty type $\bot$. A function returning—and thus proving—$\bot$ can only exist having false premises itself, as no total function without false premises can return a value in an empty codomain. The variables in a proposition correspond to polymorphic type variables. Thus, a proof of $A \to B \to A \wedge B$

[1] $p_{finite} = p_{terminating} + (1 - p_{terminating})p_{finite}^2$
$= min\{p_{finite} : 0.75p_{finite}^2 - p_{finite} + 0.25 = 0\}$[11]

```
postulate
  -- From the Haskell 'choose :: Random a => (a,a) -> Gen a'.
  choose : {{Random a}} → a × a → Gen a
```

Figure 1: The postulated choose function.

is given by:

```
proof : {A B : Set} → A → B → A × B
proof a b = (a , b)
```

"{A B : Set}" denotes that A and B are generic types, values of the 'type universe' Set, instead of being a concrete type like Nat. Note that in Agda, types themselves are first-class values [16].

A parameter wrapped in single braces makes that parameter implicitly passed to a function; its value will be inferred by Agda and does not have to be passed explicitly. A parameter wrapped in double braces, like in Figure 1, searches for an instance value in scope. Both types of arguments can still be passed a value manually by wrapping the value in single or double braces, respectively. Parameters wrapped in parentheses can name the value of a parameter for use in the rest of the type, but these must be passed explicitly and behave like a 'normal' unnamed parameter.

In names with holes, denoted by underscores, arguments can be provided in the place of the holes instead of just following the name. An example of this is the tuple type ×, which is declared as _×_, allowing it to be used as A × B instead of × A B.

Agda also supports postulates, whereby a declaration is assumed to exist but not implemented, as shown in Figure 1. The Agda compiler will treat a postulate as opaque and assume that such a function or data type can exist. This possibly loses some mathematical strictness, as now an unconstructible function, such as one proving ⊥ from truth, can be used in further proofs, possibly compromising the mathematical correctness of the entire system, similar to a contradicting set of axioms.

A limitation of proof development in programming languages is that proving even simple properties may require more effort than empirical testing. This is particularly the case when a property is itself incorrect. The development process can therefore be improved by detecting such issues early on.

2.4 AGDA2HS

AGDA2HS [4] was created to transpile Agda code into Haskell code. In contrast to Haskell, Agda lacks an extensive ecosystem of tools surrounding it, making it less accessible to the wider developer audience. AGDA2HS aims to make the mathematical security of Agda available in Haskell, allowing users both to replace existing Haskell implementations with verified Agda counterparts and to collaborate with developers who are not familiar with Agda.

Language constructs are transpiled by implementing them in Agda, annotating their definitions with a 'pragma', and invoking AGDA2HS on the Agda source files. The pragma, a compiler hint, specifies how AGDA2HS should transpile a given definition. There are multiple pragmas available for AGDA2HS, most of which follow the from:

```
{-# COMPILE AGDA2HS ... #-}
```

where the ellipsis is replaced with the command. For this paper, it is sufficient to note that there are, amongst others, AGDA2HS pragmas to generate new Haskell code and map Agda code onto existing Haskell code. Included are pragmas specifying which Haskell language structure should be targeted, such as newtype, class or data, as these can all be created from an Agda record.

Most Agda libraries are not available for use with AGDA2HS, as they cannot be properly compiled to Haskell. Instead, a translated copy of the majority of the Haskell Prelude, Haskell's standard library, is provided [4]. Furthermore, Set is renamed to Type to better align with Haskell conventions.

When a function or type defined within a /Haskell directory is used in Agda, AGDA2HS will translate it by mapping it to a Haskell function or data type of the same name and relative file path, even if it does not exist in Haskell. This constitutes an exception to the usual mechanism of generating Haskell code from Agda definitions. Furthermore, rewrite rules may be used during compilation to rename functions and data structures, which is necessary when identifiers are invalid in either language.

Finally, the @0 annotation [17; 18] marks a value as erased, indicating that it is not present at runtime. The type checker enforces that no non-erased computations depend on erased computations. This mechanism is particularly important for AGDA2HS, since it relies on erasure annotations to determine which components of an Agda program are preserved during transpilation and which parts are used solely for the Agda-side proofs. The advantage of erasure annotations is that, instead of programmers having to manually verify, Agda will ensure that the erased argument is never returned or pattern matched on, so erasing it does not affect the runtime behaviour of the program.

3 Porting QuickCheck to AGDA2HS

As discussed in the previous section, integrating QuickCheck's testing functionality into AGDA2HS can improve the developer experience by identifying bugs before a proof attempt. This section presents the approach used to expose the QuickCheck interface within Agda for use with AGDA2HS. First, the general approach to porting the library is explained in subsection 3.1. Then, subsection 3.2 details the porting of functions and data types, followed by the method for porting type classes in subsection 3.3. Finally, the possibility of showing a correspondence relation between a test and a proposition is explored in subsection 3.4.

```agda
record Gen (a : Type) : Type where
  no-eta-equality; pattern
  constructor MkGen
  field unGen : QCGen → ∃ Int IsNonNegativeInt → a

record GenN (a : Type) : Type where
  -- (...)
  field unGenN : QCGen → Nat → a
```

Figure 2: Generator definitions.

3.1 Approach

In this thesis, following the approach adopted in the AGDA2HS port of the *containers* library [19], QuickCheck is ported to Agda for AGDA2HS by postulating functions of QuickCheck in Agda, as shown in Figure 1. Where a concrete implementation is relevant for a proof and where such an implementation would allow reasoning grounded in code instead of postulates, functions may alternatively be implemented explicitly. This can improve the transparency of the proof and thus the mathematical support for the claimed property.

Additionally, all code originating from QuickCheck is placed in a `/Haskell` folder. By mirroring the structure of the QuickCheck source code and defining functions and data types in their corresponding files, calls and import statements are mapped directly onto to the existing QuickCheck library. Although additional code is introduced to the port, this approach results in that the library is *not* reimplemented but rather given an interface.

Only exported components of QuickCheck or those strictly required for the implementation of other definitions are included in the port. The functions and data types that are not exported by QuickCheck are therefore left out. Furthermore, during development, priority was given to the subset of functionality necessary for a minimal viable product.

3.2 Porting Functions and Types

The functions and data types in QuickCheck are, as described in subsection 3.1 and shown in Figure 1, primarily ported by postulating their signatures. In this example, the `choose` function is made available in Agda. It takes a 2-tuple of some type `a`, which has an instance of the `Random` type class (see subsection 3.3), and returns `Gen a`. That generator produces values between the two bounds in the tuple, although "between" is not always well-defined, as discussed in subsection 4.2.

Data types may also be postulated but are often implemented directly. Doing so allows for more transparent proofs about the nature of the data types and functions involving them. For instance, constructors and record field projections are only available when the type has an explicit implementation.

For example, generators are implemented as a record, as shown in Figure 2. As discussed in subsection 2.2, generators are functions (here named "`unGen`") that take two values: an RNG of type `QCGen` and a size parameter of type `Int`. However, in the port the size argument is wrapped in `∃ Int IsNonNegativeInt`, requiring proof that the value is non-negative. This proof is erased during transpilation. In Haskell, supplying a negative size causes QuickCheck to throw an exception. Since Agda (currently [20]) prevents all functions from throwing an exception, such calls are excluded by construction, making the non-negativity requirement explicit.

A more natural implementation of a size parameter would use `Nat` type, as all its values are non-negative. However, changing the type of a parameter in the translation step from `Nat` to `Int` would cause problems for surrounding code that still supplies or expects a `Nat`. For this reason, a second generator type, `GenN`, is introduced which transpiles to a Haskell `newtype`. With the function `toGen : GenN a → Gen a`, it can be converted to a normal generator and be used with QuickCheck. In addition, function involving a `Gen` generally have a `GenN` equivalent, distinguished by a "`N`" suffix (e.g. `chooseN`).

One of the problems of porting Haskell code to Agda is that certain Haskell names are not allowed in Agda. An example of this is the QuickCheck `.&&.` function:

```haskell
(.&&.) :: (Testable a, Testable b) =>
          a -> b -> Property
```

It takes two arguments, where both sides must implement the type class `Testable`, but can be different types, and returns a `Property` representing whether both arguments hold. The dots in the function name are not allowed in Agda, so instead for the translation, the • (U+2219: BULLET OPERATOR)[2] [21] is used in combination with rewrite rules. Now, `•&&•` will correctly be mapped to `.&&.` in Haskell. Another example is the `Discard` data type in Haskell, of which the Haskell constructor has the same name as the type. Given that in Agda, constructors cannot have the same name as a type, the constructor is named `MkDiscard` and, using rewrite rules, transpiled to the Haskell constructor `Discard`.

3.3 Porting Type Classes

Type classes in Haskell are analogous to interfaces in languages such as Java [22]. Required type classes for a type variable are specified in Agda using parameters wrapped in double braces. For example, "`{{Random a}} →`" in Figure 1 requires that `a` has an instance of the `Random` type class. This gets transpiled to "`Random a =>`" in Haskell.

[2] • is written as "`\.`" with Agda-mode [13].

```
1  record Arbitrary (a : Type) : Type where
2    no-eta-equality
3    field
4      arbitrary : Gen a
5      shrink : a → List a
6
7  open Arbitrary {{ ... }} public
8
9  {-# COMPILE AGDA2HS Arbitrary existing-class #-}
10
11 instance
12   iArbitraryCircle : Arbitrary Circle
13   iArbitraryCircle .arbitrary = MkCircle <$> arbitrary
14   iArbitraryCircle .shrink (MkCircle r) = MkCircle <$> (shrink r)
15
16 {-# COMPILE AGDA2HS iArbitraryCircle #-}
17
18 postulate
19   instance
20     iRandomBool : Random Bool
21     iRandomInt  : Random Int
```

Figure 3: Implementing a QuickCheck type class in AGDA2HS, with example instances.

All type classes in this port are implemented and thus not postulated, as an implementation of a type class is needed to access its functions or reason about its properties. Figure 3 illustrates this approach using the `Arbitrary` type class. The pragma on line 9 maps the type class onto QuickCheck's `Arbitrary` type class, even though it has a separate implementation in Agda. The 'existing-class' pragma indicates to AGDA2HS that it should transpile the record to an existing type class instead of a data type. Akin to postulating functions and data types, for this pragma to make the translation correctly, it is imperative that the names and types of the type class match up exactly to their corresponding Haskell class.

Furthermore, certain type classes must be instantiable for user-defined types. An example of this is the `Arbitrary` instance in Figure 3 of the user-defined `Circle` type. The `arbitrary` function constructs a generator for `Circle` by applying the constructor `MkCircle`—which takes a single `Nat` denoting the circle's radius—to the generator for `Nat`. Similarly, the `shrink` function, which is used by QuickCheck to shrink counterexamples, is also implemented by relying on the `shrink` implementation of `Nat`.

Nevertheless, for class instances which are already implemented in Haskell and are not implementable due to missing dependencies in Agda or for which instance implementations are not required, postulates can be used. Examples include the `Random` instances shown in Figure 3.

3.4 Testing the Correct Proposition?

When testing a proposition, it is important that the property asserted by the test is the same proposition the programmer intends to verify. Additional functionality is introduced on top of QuickCheck, using which a correspondence relation between propositions and tests can be shown.

`Reflects`[3], as shown in Figure 4, is used to express this correspondence between a proposition and a test. More precisely, it shows that for some input the test returning `True` means that the proposition holds for that input and, inversely, does not hold if the test returns `False`.

Due to limitations in the definition of `corresponds-to`, tests for which a programmer wants to show correspondence, can take one argument and must return a `Bool`. While there are equivalent functions for tests and propositions taking up to five arguments, tests requiring additional inputs can be expressed by grouping arguments into a tuple.

Currently, correspondence can only be shown for tests returning `Bool`. While a boolean result directly indicates whether a proposition holds, a `Property` may also represent a discarded test case, introducing a third outcome and substantially complicating correspondence proofs.

To facilitate these proofs, helper functions are provided that translate between boolean operators and their equivalents in the Curry–Howard correspondence. There are functions translating between "`&&`" and "×", "`||`" and "`Either`", "`not`" and "→ ⊥", "`==`" and "≡" and between "`a /= b`" and "`a ≡ b → ⊥`". The first two functions are shown in Figure 4. Note how a reversed translation is constructed by exchanging the types of the last argument and the result.

In Figure 5 an example of a correspondence proof can be seen. The proposition states that appending a list with an empty list leaves it unchanged (and thus is equivalent as applying the identity function). To ensure that the proof-in-construction on line 5 and the correspondence proof on line 11 are considering the same proposition, it is abstracted away to the `proposition-++-[]-is-id` function, though

[3] AGDA2HS's `Reflects` is analogous to the `Reflects` in the Agda standard library.

```
@0 _corresponds-to_ : (a → Type) → (a → Bool) → Type
_corresponds-to_ P pred = ∀ x → Reflects (P x) (pred x)

reflectsIsTrue : (b : Bool) → Reflects (IsTrue b) b
reflectsIsTrue True  = IsTrue.itsTrue
reflectsIsTrue False = λ ()

&&-to-× : ∀ {x y : Bool} → IsTrue (x && y) → IsTrue x × IsTrue y
&&-to-× {False} ()
&&-to-× {True} p = IsTrue.itsTrue , p

×-to-&& : ∀ {x y : Bool} → IsTrue x × IsTrue y → IsTrue (x && y)
×-to-&& {False} (() , _)
×-to-&& {True} (_ , p) = p
```

Figure 4: The framework for proofs of correspondence between propositions and tests.

```
proposition-++-[]-is-id : List a → Type
proposition-++-[]-is-id xs = xs ++ [] ≡ xs

postulate
  proof-++-[]-is-id : ∀ (xs : List a) → proposition-++-[]-is-id xs
  -- TODO proof this; test it first

prop_append_empty : List Int → Bool
prop_append_empty xs = xs ++ [] == xs

++-[]-is-id-corresponds : proposition-++-[]-is-id corresponds-to prop_append_empty
++-[]-is-id-corresponds xs = mapReflects ==-to-≡ ≡-to-==
                             (reflectsIsTrue (prop_append_empty xs))
```

Figure 5: Showing the correspondence between a proposition and a test.

they could instead be copied manually.

A difference between the proposition and the test in Figure 5 is that the proposition reasons about lists of any type, while the test reasons about lists of the `Int` type. QuickCheck replaces any unconstrained type variables with the "`()`" unit type, which contains only a single value. As a result, certain properties, such as a list staying sorted under a certain operation, cannot be meaningfully tested. Using a richer concrete type strengthens the guarantees provided by the test.

In Figure 5, correspondence is established by the implementation of the function `++-[]-is-id-corresponds`. In this example, the proof follows directly from translating the equality operator. More complex propositions, such as $a = b \wedge b \neq c$, require multiple translation steps. Using the `***` function, translations can be made in multiple steps:

```
_***_ : (a → b) → (c → d) → a × c → b × d

(λ h → (==-to-≡ *** /=-to-≡→⊥) (&&-to-× h))
(λ h → ×-to-&& ((≡-to-== *** ≡→⊥-to-/=) h))
```

For the `Either` type, the `mapBoth` function has been added to allow for analogous translation as with `***` for `×`.

4 Analysis of the Port

Given the port from section 3, I analyse it and its shortcomings and justify the chosen approach to port QuickCheck to Agda. In subsection 4.1, the current available functionality of the port is illustrated. Then, in subsection 4.2, existing problems and limitations a user of this port will encounter are discussed.

4.1 Available Functionality

The functions exposed through the process described in section 3, allow most tests to be written in AGDA2HS. For example, almost all functions related to `Property` or `Gen`, such as `==>`, `chooseInt` and `oneof` are available. A complete overview of all ported functionality is provided in Appendix A.

For the ported functions, the restrictions that are implicit in QuickCheck have been made explicit at the type level. An example is the `oneof` function, which selects a generator from a list of generators, which must be non-empty. Whereas QuickCheck enforces this by throwing an exception when encountering an empty list, the port requires proof that the list of generators is non-empty. Furthermore, properties of returned values have been encoded in the return type of certain

functions, such as bounds on generated ranges, allowing these guarantees to be used in subsequent proofs.

When such properties require stricter preconditions on functions than imposed by QuickCheck, a mirrored version is created, such as with `chooseInt` and `chooseInt'`. `chooseInt'` includes a proof that the `Int` generated will be between the two bound given by the range tuple, but requires a proof that the first value of the tuple is not larger than the second value.

The following property test can be made about the aforementioned user-made `Circle` type, which asserts that the area of a circle quadruples every time the radius is doubled:

```
prop : Circle → Property
prop c = area (scale 2 c) === 4 * area c
```

This test can then be evaluated by providing it to the `quickCheck` function. If a programmer wants to make a custom generator for a type, for example one using `GenN`, it can be done as follows:

```
circleGenN : GenN Circle
circleGenN = sizedN $ λ n →
          MkCircle <$> chooseNatN (0 , n)
```

To use this generator instead of the generator specified in the `Arbitrary` instance of `Circle`, the `forAll(N)` function can be used:

```
propN : Property
propN = forAllN circleGenN prop
```

With the added functionality to show the correspondence between tests and propositions, a programmer may be confident that they are testing the correct property. Moreover, when a proof is deferred, providing tests and a correspondence proof offers stronger evidence for correctness than relying solely on a postulate. Although translations are currently provided only for commonly used operators, users can extend them with additional translations as required.

4.2 Limitations

A user of this port may encounter two main problems, the first of which has an existing workaround. Due to a bug in AGDA2HS[4], when, for example, creating a new instance of the `Arbitrary` type class in Agda without referencing the function names elsewhere in the file, AGDA2HS will not import those functions. This causes the Haskell compiler to report errors due to unresolved identifiers. A workaround for this is to import the entirety of QuickCheck by adding a foreign pragma to the top of any Agda file involving QuickCheck with `import Test.QuickCheck` as its content, inserting the import statement into the Haskell file.

The second limitation of the port is that some parts of QuickCheck fell out of the scope for this thesis and remain unimplemented. Although this port can be extended and, in its current state, have added value to projects, approximately half of the public QuickCheck functionality is not yet available. Feature selection was prioritised according to their relevance to testing propositions, consequently leaving some areas of the QuickCheck codebase almost entirely without Agda bindings. Most notably type-level modifiers, which allow extra restrictions on the generated test data, such as all generated numbers being non-negative, are currently not supported.

When showing the relation between a property test and a proposition, the quality of the generators used is not accounted for. As discussed in subsection 2.2, a low-quality generator can have blind spots, lowering the value of the tests depending upon it. Ideally, a stronger correspondence relation would be incorporated into the port which requires a surjectivity proof for all generators used. Such proofs state that for every value of a type, a combination of an RNG seed and a size parameter exists for which the generator generates that value. Creating these proofs in this port is still an open research problem, rendering such stronger relations currently unprovable.

A further limitation of this feature is that only functions that return boolean values can be shown to correspond to a proposition. Although with these operators the majority, if not the entirety, of tests can be written, it prevents the use of the QuickCheck operators. These operators return a `Property` value and are more descriptive: where a test requires that two values are equal, the built-in == can only assert whether they are equal, while QuickCheck's === can also extract the values in case of a failed test, allowing QuickCheck to display them to the user. Additionally, the discarding functionality for inputs for which a given predicate does not hold, as implemented with `==>`, is unavailable in this context.

When porting from QuickCheck, some implicit properties cannot be properly made explicit. One such example this is the `chooseEnum` function, which takes a pair of values of a type implementing the `Enum` type class and returns a generator generating values between the first value as a lower bound and the second value as an upper bound. If the first value is greater than the second value, they are in practice swapped around by QuickCheck, although strictly speaking the behaviour is undefined. While this can be prevented by enforcing that the first value is smaller than the second value, the comparison used can have unexpected results. Enums in the context of the `chooseEnum` function are not ordered by an `Ord` implementation, but rather by their `Enum` implementation, which can have a different ordering. If one wants to make the property explicit that the first value should be smaller than the second value or that the return value will be between the two given values, the function signature either becomes cluttered with `fromEnum` calls or has them abstracted away. An alternative is requiring the `Enum` and `Ord` instances to adhere the same ordering. This limits the use of `chooseEnum` (or an alternative `chooseEnum'`) to only types for which this specific relation exists.

Moreover, making all the implicit preconditions that QuickCheck imposes explicit, several of which are technicalities, risks making the port impractical to use. If a programmer is always obliged to prove that their tests are correct before being able to test the proposition they actually wish to test, this could become a slow and 'bureaucratic' process, diminishing the incentive to test at all.

[4] `https://github.com/agda/agda2hs/issues/454`

5 Responsible Research

All code written for this project is published as a AGDA2HS fork under the MIT license[5], which means anyone can use and extend the code written for this project. This is also the license as AGDA2HS uses; thus, the code or a future version can, with relative legal ease, be merged back into AGDA2HS if the maintainers want to do so.

Although generative AI models were consulted during the writing of the code and this paper, all code and text are handwritten by me. Additionally, any relevant information brought up by an AI model has been fact-checked by me and, when used in this paper, backed with a verified citation. The AI model families consulted for this project are Mistral's *Vibe* [23] and Anthropic's *Sonnet 4.6* [24]. In addition, I used Quillbot [25] to filter out some of the grammar mistakes in this text.

By using AI minimally, the negative sides of AI usage have likewise been minimised whilst keeping the advantages. One such advantage of using AI is rapid searching of the internet or documentation, especially when, for some feature, there is no real documentation but only code examples in production code. A possible disadvantage of using AI when copying its code solutions is that the quality of the codebase decreases, more so when one does not take the time to understand its code. Additionally, two genuine concerns with AI usage are the environmental impact [26] and the possibility that it reproduces code under a restrictive license, which would result in the project containing copyright infringement. Both concerns have respectively been minimised and avoided by this approach.

6 Related Work

With the exception of one 2011 'toy' project written in Elisp [27], no testing framework is yet available in Agda 2. In 2003, Dybjer *et al.* [28] ported QuickCheck to an earlier version of Agda—Agda 1—combined with the visual proof editor Alfa, but I was not able to find the codebase accompanying their paper. In a later 2004 paper [29], they expanded their port by adding the ability to prove that a generator is surjective, that is, capable of generating all values of a given type.

Paraskevopoulou *et al.* have made a comparable effort by porting QuickCheck to the proof assistant Rocq [12]. In their version of QuickCheck, named QuickChick, functionality was added not only for surjectivity proofs of generators but also for proving that a test is equivalent to a proposition. Furthermore, as it is implemented in Rocq, most of QuickChick's codebase could be formally verified as correct.

In 2004, Berghofer *et al.* [30] introduced QuickCheck-based testing functionality to the proof assistant Isabelle/HOL, adding automatic derivation of generators for data types. In 2012, Bulwahn [31] extended Isabelle's QuickCheck version by combining random testing with exhaustive and narrowing-based testing. Exhaustive testing entails verifying a proposition for all values up to a given bound, while narrowing-based testing partially specifies values and gradually refines them, allowing it to explore many possibilities more efficiently and find counterexamples that exhaustive testing might miss.

[5] `https://github.com/MauritsLCSloof/agda2hs`

7 Discussion

Currently, the port is part of a fork of the AGDA2HS codebase akin to the aforementioned *containers* library [19]. In the future, if it is decided to extend this port and once it has matured enough, it should become a standalone project.

Alternative ways of porting QuickCheck are the approaches Paraskevopoulou *et al.* used for Rocq [12], Dybjer *et al.* [28] used for an earlier version of Agda and Berghofer for Isabelle/HOL [30], which entails reimplementing QuickCheck instead of creating an interface for it. In these projects, the generation of randomness is done outside of the proof assistant, in OCaml, Haskell and, e.g., Standard ML, respectively.

Although this method has the advantage of allowing the port of QuickCheck to be verified itself, the execution of tests would still take place in Haskell. Since the goal of this port is to expose QuickCheck specifically for AGDA2HS, it is natural to map onto the existing QuickCheck code. This way, instead of reimplementing QuickCheck in Agda to transpile it back to Haskell, the port can be relatively quickly extended by postulating QuickCheck's functions and building on top of them.

8 Conclusions and Future Work

In this thesis, I addressed the lack of testing frameworks for AGDA2HS by porting QuickCheck to Agda. Although the port is not yet complete, it has reached a state where it can be used in AGDA2HS projects. Furthermore, I extended the QuickCheck functionality to include correspondence proofs, which show that a property test and a proposition assert the same concept.

However, the correspondence proofs could be further improved as they are weak: the generator generating the test data is not required to be surjective. In the future, requiring surjectivity proofs for a generator accompanying a test would allow for a stronger correspondence relation, but a method to prove surjectivity would have to be devised first.

Additionally, the current approach to establishing correspondence imposes restrictions on the return type of the test. Methods that are capable of more expressive return types while preserving the correspondence guarantees remain an open problem for future work.

References

[1] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, p. 107–115, Jul. 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[2] W. Denman, M. H. Zaki, S. Tahar, and L. Rodrigues, "Towards flight control verification using automated theorem proving," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 89–100.

[3] N. Leveson and C. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[4] J. Cockx, O. Melkonian, L. Escot, J. Chapman, and U. Norell, "Reasonable agda is correct haskell: writing verified haskell using agda2hs," in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, ser. Haskell 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 108–122. [Online]. Available: https://doi.org/10.1145/3546189.3549920

[5] U. Norell, "Towards a practical programming language based on dependent type theory," Ph.D. dissertation, Chalmers University of Technology and Goteborg University, 2007.

[6] T. B. de la Tour and C. Kreitz, "Building proofs by analogy via the curry-howard isomorphism," in *Logic Programming and Automated Reasoning*, A. Voronkov, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 202–213.

[7] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 268–279. [Online]. Available: https://doi.org/10.1145/351240.351266

[8] S. Juhošová, "How novices perceive interactive theorem provers (extended abstract)," in *Proceedings of Workshop on Type-Driven Development*, ser. TyDe '24. Milan, Italy: Association for Computing Machinery, 2024, new York, NY. [Online]. Available: https://icfp24.sigplan.org/details/tyde-2024-papers/1/How-Novices-Perceive-Interactive-Theorem-Provers-Extended-Abstract-

[9] G. Hutton, *Programming in haskell*. Cambridge University Press, 2016.

[10] K. Claessen, "Quickcheck: Automatic testing of haskell programs," https://hackage.haskell.org/package/QuickCheck, 2007 (accessed May 29, 2026).

[11] K. B. Athreya and P. E. Ney, *Branching processes*. Springer Science & Business Media, 2012.

[12] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce, "Foundational property-based testing," in *International Conference on Interactive Theorem Proving*. Springer, 2015, pp. 325–343.

[13] "Welcome to Agda's documentation!" https://agda.readthedocs.io/en/stable/, 2025 (accessed May 1, 2026).

[14] Z. Luo, *Computation and reasoning*. Oxford University Press Oxford, 1994, vol. 49.

[15] P. Martin-Löf, "An intuitionistic theory of types: Predicative part," in *Logic Colloquium '73*, ser. Studies in Logic and the Foundations of Mathematics, H. Rose and J. Shepherdson, Eds. Elsevier, 1975, vol. 80, pp. 73–118. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0049237X08719451

[16] J. Cockx, "Programming and proving in agda," https://github.com/jespercockx/agda-lecture-notes, 2026 (accessed May 29, 2026).

[17] R. Atkey, "Syntax and semantics of quantitative type theory," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 56–65. [Online]. Available: https://doi.org/10.1145/3209108.3209189

[18] C. McBride, "I got plenty o'nuttin'," in *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, 2016, pp. 207–233.

[19] H. Apfelmus, "Agda2hs containers library," https://github.com/agda/agda2hs/tree/master/lib/containers, 2025 (accessed May 22, 2026).

[20] J. Cockx, "Add basic support for exceptions," https://github.com/agda/agda2hs/pull/445, 2025 (accessed Jun. 8, 2026).

[21] "Mathematical operators." https://www.unicode.org/charts/PDF/U2200.pdf, (accessed May 27, 2026).

[22] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 60–76.

[23] "Mistral vibe (formerly le chat) - ai chat and coding agent," https://mistral.ai/products/vibe/, 2026 (accessed Jun. 8, 2026).

[24] "Anthropic," https://www.anthropic.com/, 2024 (accessed Jun. 8, 2026).

[25] "Quillbot: Write, design & create," https://quillbot.com/, 2017 (accessed Jun. 8 2026).

[26] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. Aga, J. Huang, C. Bai *et al.*, "Sustainable ai: Environmental implications, challenges and opportunities," *Proceedings of machine learning and systems*, vol. 4, pp. 795–813, 2022.

[27] "Unit testing agda code with elisp," https://pozorvlak.livejournal.com/160409.html, 2011 (accessed Jun. 1, 2026).

[28] P. Dybjer, Q. Haiyan, and M. Takeyama, "Combining testing and proving in dependent type theory," vol. 2758, 06 2003.

[29] ——, "Random generators for dependent types," in *International Colloquium on Theoretical Aspects of Computing*, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:13634329

[30] S. Berghofer and T. Nipkow, "Random testing in isabelle/hol." in *SEFM*, vol. 4, 2004, pp. 230–239.

[31] L. Bulwahn, "The new quickcheck for isabelle: Random, exhaustive and symbolic testing under one roof," in *International Conference on Certified Programs and Proofs*. Springer, 2012, pp. 92–108.

A List of Functions, Data Types and Type Classes

This appendix contains a list of all functions, data types and type classes available to users of the port. Functions and types that do not originate from the Haskell QuickCheck library are presented separately. For some groups of functions with long yet similar function signatures, the common part of the function signature is abstracted away to a `module _ ... where` block to increase readability.

A.1 Functions

From QuickCheck

```
sized : (∃ Int IsNonNegativeInt → Gen a) → Gen a
getSize : Gen (∃ Int IsNonNegativeInt)
resize : ∃ Int IsNonNegativeInt → Gen a → Gen a
scale : (∃ Int IsNonNegativeInt → ∃ Int IsNonNegativeInt) → Gen a → Gen a
choose : {{Random a}} → a × a → Gen a
chooseAny : {{Random a}} → Gen a
chooseInt : (range : Int × Int) → Gen Int
chooseEnum : {{Enum a}} → (range : a × a) → Gen a
chooseInteger : (range : Integer × Integer) → Gen Integer
generate : Gen a → IO a
sample' : Gen a → IO (List a)
sample : {{Show a}} → Gen a → IO ⊤
genDouble : Gen (∃ Double (0.0 <=_<= 1.0))
suchThat : Gen a → (P : a → Bool) → Gen (∃ a (λ x → IsTrue (P x)))
suchThatMap : Gen a → (a → Maybe b) → Gen b
suchThatMaybe : Gen a → (P : a → Bool) → Gen (Maybe (∃ a (λ x → IsTrue (P x))))
oneof : ∃ (List (Gen a)) NonEmpty → Gen a
frequency : ∃ (List ((∃ Int IsNonNegativeInt) × Gen a)) NonEmpty → Gen a
elements : ∃ (List a) NonEmpty → Gen a
sublistOf : List a → Gen (List a)
shuffle : List a → Gen (List a)
growingElements : ∃ (List a) NonEmpty → Gen a
listOf : Gen a → Gen (List a)
listOf1 : Gen a → Gen (List a)
vectorOf : Int → Gen a → Gen (List a)
ioProperty : IO prop → Property
idempotentIOProperty : IO prop → Property
module _ {prop : Type} {{_ : Testable prop}} where
  forAll : {{Show a}} → Gen a → (a → prop) → Property
  forAllShow : Gen a → (a → String) → (a → prop) → Property
  forAllBlind : Gen a → (a → prop) → Property
  forAllShrink : {{Show a}} → Gen a → (a → List a) → (a → prop) → Property
  forAllShrinkShow : {{Show a}} → Gen a → (a → List a) → (a → String) →
                                                    (a → prop) → Property
  forAllShrinkBlind : {{Show a}} → Gen a → (a → List a) → (a → prop) → Property
_==>_ : {{Testable a}} → Bool → a → Property
_===_ : {{Eq a}} → {{Show a}} → a → a → Property
_=/=_ : {{Eq a}} → {{Show a}} → a → a → Property
_•&•_ : {{Testable a}} → {{Testable b}} → a → b → Property
_•&&•_ : {{Testable a}} → {{Testable b}} → a → b → Property
_•||•_ : {{Testable a}} → {{Testable b}} → a → b → Property
quickCheck : {{Testable a}} → a → IO ⊤
```

New

```
@0 _<=_<=_ : {{Ord a}} → a → a → a → Type
chooseInt' : (range : Int × Int) → @0 {{IsTrue (fst range <= snd range)}} →
                                          Gen (∃ Int (fst range <=_<= snd range))
chooseInteger' : (range : Integer × Integer) → @0 {{IsTrue (fst range <= snd range)}} →
                                          Gen (∃ Integer (fst range <=_<= snd range))
```

```
toGen : GenN a → Gen a
toGenN : Gen a → GenN a
sizedN : (Nat → GenN a) → GenN a
getSizeN : GenN Nat
resizeN : Nat → GenN a → GenN a
scaleN : (Nat → Nat) → GenN a → GenN a
chooseN : {{Random a}} → a × a → GenN a
chooseAnyN : {{Random a}} → GenN a
chooseIntN : (range : Int × Int) → GenN Int
chooseIntN' : (range : Int × Int) → @0 {{IsTrue (fst range <= snd range)}} →
                                                GenN (∃ Int (fst range <=_<= snd range))
chooseNat : (range : Nat × Nat) → Gen Nat
chooseNatN : (range : Nat × Nat) → GenN Nat
chooseEnumN : {{Enum a}} → (range : a × a) → GenN a
chooseIntegerN : (range : Integer × Integer) → GenN Integer
chooseIntegerN' : (range : Integer × Integer) → @0 {{IsTrue (fst range <= snd range)}} →
                                            GenN (∃ Integer (fst range <=_<= snd range))
generateN : GenN a → IO a
sampleN' : GenN a → IO (List a)
sampleN : {{Show a}} → GenN a → IO ⊤
genDoubleN : GenN (∃ Double (0.0 <=_<= 1.0))
suchThatN : GenN a → (P : a → Bool) → GenN (∃ a (λ x → IsTrue (P x)))
suchThatMapN : GenN a → (a → Maybe b) → GenN b
suchThatMaybeN : GenN a → (P : a → Bool) → GenN (Maybe (∃ a (λ x → IsTrue (P x))))
oneofN : ∃ (List (GenN a)) NonEmpty → GenN a
frequencyN : ∃ (List (Nat × GenN a)) NonEmpty → GenN a
elementsN : ∃ (List a) NonEmpty → GenN a
sublistOfN : List a → GenN (List a)
shuffleN : List a → GenN (List a)
growingElementsN : ∃ (List a) NonEmpty → GenN a
listOfN : GenN a → GenN (List a)
listOf1N : GenN a → GenN (List a)
vectorOfN : Int → GenN a → GenN (List a)
module _ {prop : Type} {{_ : Testable prop}} where
  forAllN : {{Show a}} → GenN a → (a → prop) → Property
  forAllShowN : GenN a → (a → String) → (a → prop) → Property
  forAllBlindN : GenN a → (a → prop) → Property
  forAllShrinkN : {{Show a}} → GenN a → (a → List a) → (a → prop) → Property
  forAllShrinkShowN : {{Show a}} → GenN a → (a → List a) → (a → String) →
                                                          (a → prop) → Property
  forAllShrinkBlindN : {{Show a}} → GenN a → (a → List a) → (a → prop) → Property
@0 _corresponds-to_ : (a → Type) → (a → Bool) → Type
@0 _corresponds-to-2_ : (a → b → Type) → (a → b → Bool) → Type
@0 _corresponds-to-3_ : (a → b → c → Type) → (a → b → c → Bool) → Type
@0 _corresponds-to-4_ : (a → b → c → d → Type) → (a → b → c → d → Bool) → Type
@0 _corresponds-to-5_ : (a → b → c → d → e → Type) → (a → b → c → d → e → Bool) → Type
reflectsIsTrue : (b : Bool) → Reflects (IsTrue b) b
not-to-→⊥ : ∀ {x : Bool} → IsTrue (not x) → (IsTrue x → ⊥)
→⊥-to-not : ∀ {x : Bool} → (IsTrue x → ⊥) → IsTrue (not x)
&&-to-× : ∀ {x y : Bool} → IsTrue (x && y) → IsTrue x × IsTrue y
×-to-&& : ∀ {x y : Bool} → IsTrue x × IsTrue y → IsTrue (x && y)
||-to-Either : ∀ {x y : Bool} → IsTrue (x || y) → Either (IsTrue x) (IsTrue y)
Either-to-|| : ∀ {x y : Bool} → Either (IsTrue x) (IsTrue y) → IsTrue (x || y)
module _ {prop : Type} {{_ : Testable prop}} where
  ==-to-≡ : ∀ {x y : ty} → IsTrue (x == y) → x ≡ y
  ≡-to-== : ∀ {x y : ty} → x ≡ y → IsTrue (x == y)
  /=-to-≡→⊥ : ∀ {x y : ty} →  IsTrue (x /= y) → ((x ≡ y) → ⊥)
  ≡→⊥-to-/= : ∀{x y : ty} → ((x ≡ y) → ⊥) → IsTrue (x /= y)
mapBoth : (a → c) → (b → d) → Either a b → Either c d
```

A.2 Data Types

From QuickCheck

```
Gen a
Result
Property
Discard
QCGen
Prop
Rose a
Callback
CallbackKind
```

New

```
GenN a
```

A.3 Type Classes

From QuickCheck

```
Random
Arbitrary
Testable
```